

---

## Problem Set 1 Solutions

### Problem 1-1. PGP

This problem has both individual and group elements to it. Your group should turn in one write-up answering each of the parts labeled **[Group]**, but all key pairs, emails, etc. should be created and sent individually.

Read Alma Whitten's paper, "*Why Johnny Can't Encrypt*." (There is a link to it on the "Lectures and Handouts" page of the course website.)

Locate and install a fresh version of PGP or GPG. There are versions for Unix flavors, Windows, and the Macintosh. See <http://web.mit.edu/network/pgp.html> for downloads. If for some reason you are not able to download from this site, <http://www.pgpi.org/> may be of use.

Find the PGP public keys for as many of the 6.857 staff as you can. Part of your assignment is figuring out how to locate PGP keys. Searching the Internet for PGP key servers may be of help. But beware; there may be fake keys out there...

- (a) **[Group] Trust and Signing.** PGP's "web of trust" model allows users to "sign" each others' public keys. Suppose Alice signs Bob's key; what, in effect, is Alice declaring when she does this? Why is it useful for people to sign each other's keys? What precautions should one take before signing someone else's key, and why are these measures appropriate? If Bob subsequently signs Charlie's key, what (if anything) does Alice, or one who personally knows and trusts Alice, know about Charlie?
- (b) **[Individual] Getting started.** Create a new public/private key pair for yourself (you may use an existing key pair if you already have one). Sign each of your group members' public keys, and have them sign yours.

When all of your group members have signed your public key, email it to [6857-submit@csail.mit.edu](mailto:6857-submit@csail.mit.edu) in ASCII-armored format, with the subject **My public key**.

- (c) **[Individual] Encrypting email.** Send an encrypted, signed email to [6857-submit@csail.mit.edu](mailto:6857-submit@csail.mit.edu) with the subject **PGP is fun**. Do *not* send the mail to the TAs individually. In the body of the message,
- Tell us what operating system, mail client, and version of PGP you are using.
  - Show us the public keys you found for the 6.857 staff; PGP fingerprints are sufficient.
  - Search for keys for [president@whitehouse.gov](mailto:president@whitehouse.gov) on a PGP key server. Based on your findings, explain (in a few sentences each) one useful feature and one drawback of PGP key servers. (You do not need to include any key(s) you found.) Remember to cite all your sources.
  - In a few sentences, explain why you do or do not believe that the keys you found for the 6.857 staff in fact belong to us. If you do not trust a public key, explain what would convince you otherwise.

Your mail should be protected with PGP such that the 6.857 TAs, and *only* the 6.857 TAs, can obtain the plaintext contents. You must also sign the mail with your private key. We will only accept your first message, so make sure to get it right the first time. Are you able to finish the assignment in fewer than 90 minutes as in Whitten's experiment? Remember to cite all your sources (books, manuals, friends, etc.) according to the guidelines in the course information handout.

**Solution:** (Thanks to Hooman Katirai, Umberto Malesci, Oleg Shamovsky, Armando Valdes)

Suppose Alice signs Bob's key; what in effect, is Alice declaring when she does this?

When attempting to sign another person's key, a PGP user is presented with the following guidance:

“By signing the selected user id(s) you are certifying based on your own direct first-hand knowledge that the key(s) and attached user id(s) actually belong to the identified user(s). Before signing, make sure the key(s) were delivered to you in a secure manner by the owner or you have verified the fingerprint with the owner.”

Under the above standard, when Alice signs Bob's key, she is vouching for the key's authenticity. That is, she is indicating her belief that the key bearing Bob's name, does in fact belong to, and is associated with, Bob. How she came to this belief and whether or not this belief is justified, is something that one cannot infer from Alice's signature alone. Of course, if one personally knows Alice and the standards she is likely to apply prior to granting a signature one could make some inferences about the degree to which her signature can be interpreted as a sign of the key's authenticity. The latter example highlights the difference between two important but distinct concepts: authenticity and trust. Authenticity may be defined as the degree to which a key can be believed to in fact belong to the person it is associated with while trust may be regarded as the degree to which a person's signature on a key is a sign of a key's authenticity. The PGP standard presented above only discusses authenticity. These two concepts are distinct. Suppose Bob is known to willingly signs bogus keys. In this case, Alice would still be justified in signing Bob's public key (because the key does in fact belong to Bob) even though she doesn't trust Bob. On the other hand if Alice knows Bob to be trustworthy, she can use this trust to infer the authenticity of keys signed by Bob.

Why is it useful for people to sign each other's keys?

Webs of trust are useful insofar as they enable people who trust each other to regard keys signed by each other as authentic. We illustrate this by example. Suppose Alice trusts Bob to only sign keys, he believes are authentic. Better yet, suppose that Alice is familiar with the procedures and standards that Bob applies before affixing his signature to a key, and that she is satisfied that such procedures are at least as rigorous as the standards she herself would apply when determining if a key was authentic. In such a scenario, if Alice trusts Bob, she can regard the keys signed by Bob as authentic.

What precautions should someone take before signing someone else's keys, and why are these measures appropriate?

Before signing someone else's key one should verify that the key does in fact belong to the name or alias associated with the key and that the key has not been altered, modified in transit, or swapped with a key belonging to someone else. This can be accomplished if the key is delivered through a secure, trusted channel such as hand-delivery from the key-holder to the key-signer. Verification of the key can be accomplished by sending the key (a hash of the key) through an alternate secure channel such as hand delivery or telephone. Some services for example, contact the key-holder by mail, or phone and quiz the key-holder with a series of questions based on their credit records, that would be difficult for people other than the legitimate key-holder to answer. Such measures are appropriate insofar as they prevent adversaries from impersonating the identity of other individuals.

If Bob subsequently signs Charlie's key, what (if anything) does Alice, or someone who personally knows and trusts Alice, know about Charlie?

There are two principles relevant here. First, signatures are not a sign of trust; rather they are a declaration of authenticity. Thus, if Alice signs a key bearing Bob's identity, she is merely declaring her belief that the key does in fact belong to Bob; however, her signature is not a statement for or against Bob's trustworthiness. As a result, one cannot infer trust from a signature. Nonetheless, Alice may have an opinion concerning Bob's trustworthiness either from her personal interactions with Bob or through her knowledge of Bob's reputation. If Alice regards Bob as trustworthy, then she can use this trust to infer the authenticity of Charlie's key, from Bob's signature of Charlie's key. Conversely, if she does not trust Bob, then Alice will not be able to conclusively authenticate Charlie's key based on Bob's signature of Charlie's key alone. The second relevant principle is that trust is not transitive, and that trust often cannot survive more than “1 hop.” That is, people who know and trust Alice and don't know Bob and are in no position to know about Bob's reputation either as a trustworthy or untrustworthy person. Lacking knowledge of Bob's trustworthiness, people who know and trust Alice and will not be able to infer anything about the authenticity of Charlie's

key, based on Bob's signature of Charlie's key. Consider, for example, the case where Bob is an untrustworthy person who signs bogus keys, such as `president@whitehouse.gov` key belonging to Cody. Even though such a public key is most certainly bogus. Even in this case, Alice would be justified in signing Bob's key, despite Bob's malicious nature. However, one could not infer the authenticity of a key from Bob's signature. Hence, lacking knowledge of Bob's trustworthiness, people who know and trust Alice could not infer the authenticity of Charlie's key based on Bob's signature even if Bob's key was signed by Alice.

### Problem 1-2. Open Source Security Model [Group]

In this problem, you will compare how an open source or proprietary model affects the number of security bugs in a piece of software. To do so, you will use a very simple software development model.

In this model, you will consider a piece of software with a fixed number of lines of code  $L$ . Software developers replace code at a fixed rate of  $\rho$  lines per time unit. A fraction  $\delta$  of new lines will contain a defect, or bug. Each bug is assumed to be on a single line and represents a single security vulnerability. At a given time  $t$ , the value  $V(t)$  represents the number of vulnerabilities currently in the code. Replaced lines are randomly chosen and may contain bugs proportional to  $V(t)$ .

In the proprietary model, a team of debuggers (the "good guys") will check lines of code at a fixed rate of  $\hat{\gamma}$  lines per time unit, and will fix all bugs present in those lines. Meanwhile, adversarial "bad guys" will check the code for security exploits at a rate of  $\hat{\beta}$  lines per time unit. Once an adversary exploits a bug, the bug will be fixed. Assume that adversaries and debuggers never simultaneously discover the same bug. The number of bugs that debuggers and adversaries find will be proportional to  $V(t)$ .

In the open source model, debuggers and adversaries can analyze code at fixed rates of  $\gamma$  and  $\beta$  lines per time unit, respectively. Since everyone has easier access to the source code, you may assume that  $\gamma > \hat{\gamma}$  and  $\beta > \hat{\beta}$ . You may also assume that  $V(0) = \delta L$ . To summarize the model:

- $L$  = Total lines of code.
- $\rho$  = Rate of line replacement.
- $\delta$  = Density of bugs in replaced line.
- $V(t)$  = Total number of vulnerabilities at time  $t$ .
- $\gamma$  = Debugger inspection rate in the open source model.
- $\hat{\gamma}$  = Debugger inspection rate in the proprietary model.
- $\beta$  = Adversarial inspection rate in the open source model.
- $\hat{\beta}$  = Adversarial inspection rate in the proprietary model.

- (a) What is the net change in bugs per time unit due to the replacement of lines of code? Your answer should be independent of debuggers or adversaries, and may depend on  $V(t)$ .

**Solution:** There are  $\rho$  new lines introduced per time unit, and they will have  $\delta\rho$  new bugs. However, the replaced lines contain  $\rho\frac{V(t)}{L}$  of the  $V(t)$  bugs currently in the code. Therefore, there is a net change of  $\rho\left(\delta - \frac{V(t)}{L}\right)$  bugs per time unit.

- (b) How many security bugs can a proprietary adversary exploit per time unit? How many can an open source adversary exploit per time unit? Your answer may depend on  $V(t)$ .

**Solution:** Since the proprietary adversary checks  $\hat{\beta}$  lines per time unit, and the density of bugs per line is  $\frac{V(t)}{L}$ , she is able to exploit  $\hat{\beta}\frac{V(t)}{L}$  bugs per time unit. Similarly, an open source adversary can exploit  $\beta\frac{V(t)}{L}$  bugs per time unit.

- (c) Find the number of vulnerabilities  $V^*$  that the system converges to in the open source model. In other words, find a value  $V^*$  such that if  $V(t) = V^*$ , then the expected value  $V(t + 1) = V^*$ .

**Solution:** We can look at how the number of vulnerabilities changes over time. Each year  $\rho(\delta - \frac{V}{L})$ , bugs result from the code update and  $(\gamma + \beta)\frac{V}{L}$  bugs are removed through debugging or exploits. Solving the differential equation can find the value  $V$  where the system does not change:

$$\begin{aligned}\frac{dV}{dT} &= \rho\delta - \frac{\rho V^*}{L} - (\gamma + \beta)\frac{V^*}{L} \\ 0 &= \rho\delta - (\rho + \gamma + \beta)\frac{V^*}{L} \\ V^* &= \frac{\rho\delta L}{\rho + \gamma + \beta}\end{aligned}$$

- (d) Determine a relationship between the rates  $\rho, \gamma, \beta, \hat{\gamma}$ , and  $\hat{\beta}$  such that there are fewer expected security exploits in the open source than in the proprietary model. You may assume the number of vulnerabilities have converged to  $V^*$  and  $\hat{V}^*$ , respectively.

**Solution:**

We want to find a relationship that satisfies  $\beta\frac{V^*}{L} < \hat{\beta}\frac{\hat{V}^*}{L}$ :

$$\begin{aligned}\beta\frac{V^*}{L} &< \hat{\beta}\frac{\hat{V}^*}{L} \\ \beta\left(\frac{\rho\delta}{\rho + \gamma + \beta}\right) &< \hat{\beta}\left(\frac{\rho\delta}{\rho + \hat{\gamma} + \hat{\beta}}\right) \\ \beta(\rho + \hat{\gamma} + \hat{\beta}) &< \hat{\beta}(\rho + \gamma + \beta) \\ \beta(\rho + \hat{\gamma}) &< \hat{\beta}(\rho + \gamma) \\ \left(\frac{\beta}{\hat{\beta}}\right) &< \left(\frac{\rho + \gamma}{\rho + \hat{\gamma}}\right)\end{aligned}$$

Thus, if  $\left(\frac{\beta}{\hat{\beta}}\right)$  is less than  $\left(\frac{\rho + \gamma}{\rho + \hat{\gamma}}\right)$ , then the open source model will have fewer bug exploits.

- (e) This model of software development is obviously very limited. Read Ross Anderson's "*Security in Open versus Closed Systems - The Dance of Boltzmann, Coase and Moore*", available on the course web page and online at: <http://www.cl.cam.ac.uk/ftp/users/rja14/toulouse.pdf>. Which aspects of software development does Anderson's model capture that this model does not? What aspects of software development and debugging do you think Anderson's model lacks?

**Solution:** (Thanks to Tony Lim, Akshay Patil, Kathryn Shih, and Tawanda Sibanda)

Aspects of software development that Anderson's model captures that this model does not:

Anderson presents a more plausible model of human bug discovery because it takes into account differences in environments and human motivation. For example, he acknowledges that the probability of detecting bugs decreases with time, whereas the model presented does not take time into account. He acknowledges that beta testers and adversaries can find bugs at the same rate. In a closed system, both the testers and adversaries are equally hampered. In the model presented, the only assumption

is that tester and adversary inspection rates decrease in the closed system. The rate of decrease does not have to be the same.

He also takes into account the different roles that exist in software development such as beta testers and alpha developers. The model in question 2 fails to take this into account.

Anderson recognizes that there is a distribution of interests in debugging where certain sections of the code get greater scrutiny. The other model assumes that there is an equal chance of each line of the code being debugged.

Aspects of software development and debugging that Anderson's model lacks:

He neither mentions nor considers the fact that one can introduce bugs in the process of debugging.

There is far more free open code than free closed code and he abstracts over it to prevent a debate. However, this is important because of issues such as code reuse.

He does not consider personal vendettas or personal ideologies that motivate individuals to target the code or programs of a specific company.

Anderson fails to consider that the level of commitment to the user differs from a proprietary model (which is often the case for closed systems) to an open source model. In the proprietary model, there is a greater degree of commitment to fix the code in response to pressure from the user when the latter finds bugs.

He does not consider the programming language that is used. Some languages are more vulnerable to bugs than others. In open source systems, common languages that are more robust tend to be used.

Anderson does not take into account the fact that in closed systems, communication between developers is easier to established relative to an open system.

The different education and experience levels of the developers and testers in both systems is not factored into the analysis.

### Problem 1-3. Vulnerability/Mechanism Chains [Group]

Consider some system  $S$  (such as a voting system) with an associated security policy  $P$  consisting of several parts  $(P_1, P_2, \dots, P_n)$ .

There may be a vulnerability such that  $P_i$  is violated by an adversary, who may be a participant. To address the violation, a new security mechanism is added to  $S$ . This new mechanism may be for *protection* against the risk that the vulnerability is exploited, or may be for *detection* that the vulnerability has actually been exploited (in which case some additional *recovery mechanism* may also be required, unless the detection mechanism is included only for its deterrence effect).

For example, there may be a vulnerability in a voting system where a voter casts more than one vote. So, the initial voting system is augmented by adding a protective mechanism  $M_1$  involving smart-cards:

**Mechanism 1** *The voter is given one smart card when she identifies herself at the polling site, and the voting terminal only allows each card to be used once. This may be implemented by erasing the card contents once the voter has voted.*

But the newly added protection, detection, or recovery mechanism  $M_1$  may also introduce new violations of the security policy policy: it may be defeated by an adversary so that  $M_1$  doesn't protect, detect, or recover as it should. So a new security mechanism  $M_2$  is added to help protect against, detect, or recover from attacks exploiting this new vulnerability in  $M_1$ .

For example, with  $M_1$  there may be a vulnerability that allows a dishonest voter to forge the appropriate smart cards (see the Kohno et. al paper, "*Analysis of an Electronic Voting System*", for example). So, an additional prevention mechanism  $M_2$  may be added to prevent forgery. (Note that this is intended to be a chain, so we are talking about prevention of card forgery now only, not prevention more generally of multiple voting.)

To prevent forgeries, we can add a public key cryptographic mechanism. Each voting terminal may be pre-loaded with list of public keys. These public keys will correspond to a secret key stored on each smart card that may be used at that polling site. A challenge-response mechanism is used at the voting terminal to verify that the card possesses the secret key corresponding to one of the authorized public keys. Once the card is used to vote, its corresponding public key is marked as “used” and cannot be used for other votes.

There may be new vulnerabilities with this countermeasure, e.g. a valid card may be still be duplicated and used on different machines. So a detection mechanism  $M_3$  is added comparing the list of used public keys at each machine at the end of the voting day, etc. In this way, chains of vulnerabilities and security mechanisms can be built up.

Read the Kohno et al. paper, or any other paper or web site about voting systems, and describe the longest such chain that you can that seems consistent with how voting is actually done today on some voting system. Don't bother to report a chain longer than eight. (By then you've got the point!)

Here are some potentially useful links:

<http://theory.lcs.mit.edu/~rivest/voting/index.html>  
<http://avirubin.com/vote/analysis/index.html>  
<http://www.csl.sri.com/users/neumann/book-voting.html>

**Solution:** (Thanks to Edmund Kay, Jelani Nelson, Brian Wu, and Vincent Yeung)

- 1.The state of Massachusetts requires voters to present name and address at polling booths to enforce voter authenticity (<http://www.sec.state.ma.us/ele/eleafv/howvote.htm>)
- 2.Mechanism 1 is flawed since someone can impersonate another person at the voting booth too easily by simply saying their name and address. We remedy this by requiring government issued ID when voting.
- 3.People can forge counterfeit government IDs to impersonate someone else, so we furthermore add a biometric test such as fingerprint scanning to ensure more security for voter authenticity. Fingerprint data is kept on a central server, and when a voter scans their fingerprint the data is sent to the server to be verified with what is in the database.
- 4.The transmission between the voting booth and central data server described in Mechanism 3 could be subject to a man in the middle attack, e.g. an adversary can impersonate the server and verify incorrect fingerprint scans or deny legitimate voters the right to vote. We solve this by using a signing/verification scheme between the server and the machines at the voting booths. We also encrypt all data to assure voter privacy concerning biometric data.
- 5.Data on the server may be modified by the adversary, so we make backups of hashes of files each night. The hash function is one way to ensure privacy of voter data, and it is also weak collision resistant so that an adversary may not replace a file while still maintaining the same hash.
- 6.Mechanism 5 only provides detection, so to add preemptive protection we require staff to have usernames and passwords to log into the data server.
- 7.Bad passwords may be subject to dictionary attacks, so to prevent this we suspend accounts temporarily which have had successive failed login attempts.
- 8.Mechanism 7 enables denial of service attacks by an adversary performing bad logins intentionally to suspend a particular person's account. We remedy this by keeping system administrators who staff can call if they feel their accounts are experiencing problems.